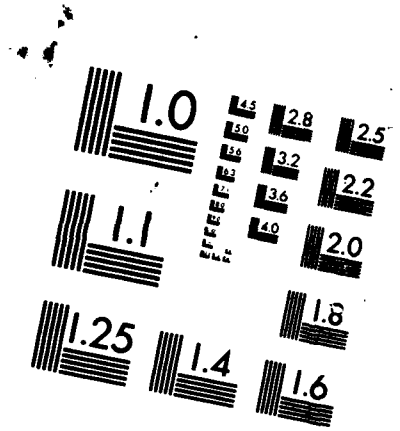
AD-A165 336 METRICS FOR ADA PACKAGES: AN INITIAL STUDY(U) MARYLAND 1/1
UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE
J D GANNON ET AL. 1985 N00014-82-K-0225

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A165 336

Metrics for Ada Packages: An Initial Study

J.D. Gannon, E.E. Katz, and V.R. Basili

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

DTIC ELECTE MAR 0 6 1986 D
E

Abstract

Many novel features of Ada (e.g., packages, generics, tasking, and exception handling) were included in the language to improve readability, aid modification, or encourage reusability. Since they have not been available in other widely used languages, packages present programmers with a fairly formidable learning task. We studied four first-time Ada programmers as they developed a ground-support satellite system. Metrics are presented which characterize their use of Ada packages, indicating where program structure may make changes difficult, and suggesting how the structure may be improved. Our findings suggest that a good background in the software engineering practices supported by Ada is necessary to learn to use the features of the language -- simply teaching professional programmers Ada is not enough.

Keywords: Ada, packages, design metrics, case studies

DTIC FILE COPY

This work is supported by the Office of Naval Research and the Ada Joint Program Office under grant N00014-82-K-0225.
Ada is a registered trademark of the U.S. Department of Defense - AJPO.

This document has been approved for public release and sale; its distribution is unlimited.

86 3 4 10

1. Introduction

When programmers begin to learn a new language, they often start by using those features of the language that appeared in other languages they already know. For example, a study of 102,397 statements in PL/I programs turned up 7385 DO statements, but only 11 DO WHILE statements [Elshoff 76]. Elshoff concluded:

> "The most basic, general form of the DO statement is not used. The fact that the programmers do not know of its existence is a primary reason."

Error messages associated with novel constructs often provoke inappropriate responses. In a language in which programmers had to explicitly request the right to access identifiers declared in outer units of scope instead of inheriting them automatically [Gannon and Horning 75], programmers responded to messages about lack of access to an identifier by making all possible identifiers visible.

2. Modules

Modules allow programmers to group related data and/or procedures and to limit the amount of information that is accessible to the rest of the program [Parnas 71]. Splitting a program into modules should localize the effects of program changes to correct errors or to improve the implementation (i.e., making it more robust or more efficient). In addition, since modules are usually self-contained, they can be reused from project to project. The designers of Ada [Ichbiah et al. 79] recognized three major uses for modules:

(1) A named collection of declarations that makes a group of types and variables available much like a FORTRAN common block.

(2) A group of related subprograms that provides a library facility.

(3) An encapsulated data type that provides the names of the type and its operations, but hides the details of the representation of objects of the type and of the implementation of the type's operations.

While the first two uses are familiar to many programmers, the third use is not supported by many commonly-used programming languages. Strong syntactic clues are available to help programmers decide what objects comprise the first two kinds of modules (e.g., all types and constants, a collection of global variables, or a set of utility routines), but fewer hints are available to aid in grouping objects in problem-oriented terms [Ledgard 85].

One of the lessons of structured programming is that simply providing users with "goto-less" programming languages does not result in structured programs being written. Users need to understand the ideas of top-down development and stepwise refinement to produce structured programs. Our hypothesis was that a similar phenomenon was likely to occur with Ada packages. Users who were unfamiliar with the ideas of information hiding and data abstraction were unlikely to use Ada packages to write programs that exhibited these properties. Thus, the need for training at the professional level would likely be for software design techniques, not just the Ada programming language.

3. Ada Packages and Units

In Ada, modules are implemented as packages. A package consists of two parts: a specification and a body. The specification, which contains declarations, is further divided into visible and private parts. Identifiers declared in the visible part can be used by other units, while those declared in the private part can only be used in the package body. For example, the following package specification exports the name of the type Rational with operations /, +, etc. The representation of a rational number by a record containing two integers is hidden from users in a private part.

```
package Rational is -- specification part
  -- visible part
  type Rational is private;
  function "/" (X,Y: integer) return Rational; -- to construct a rational
  function "+" (X,Y: Rational) return Rational;
  ...
private
  type Rational is
    record
      Numerator, Denominator: integer;
    end record;
end;
```

The package body contains implementations of operations and declarations of types whose names appear in the corresponding specification part. Nothing declared in the package body is visible outside the package. However, package bodies and specifications can use information from other packages' specifications.

Packages are not required to hide the representation of data. The specification for Rational numbers above could have been declared as follows.

```
package Rational is -- specification part
  -- visible part
  type Rational is
    record
      Numerator, Denominator: integer;
    end record;
  function "/" (X,Y: integer) return Rational; -- to construct a rational
  function "+" (X,Y: Rational) return Rational;
  ...
end;
```

However, if the representation is defined in the visible part of the specification, any other units which can see the package can manipulate the representation of the data (e.g., may access the Numerator or Denominator field of any Rational object). Changes in the representation, therefore, might have a tremendous effect on those units.

Encapsulating data types in packages allows the definition of objects and their associated operations. Hiding the representation of the data either in the private part of the package specification or in the package body limits the effects of changes in the representation. If it is in the private part, changing the representation can necessitate recompilation of units using the package. If it is in the body, changes will not force recompilation, but the implementation may be more difficult.

Ada programs are collections of compilation units: pre-defined units, package specifications, and others (i.e., subprogram, package, and task bodies). Where a package is defined is another important decision. A package might be generally available to any other unit in the environment. It may be defined in a library restricted to a project or group which will limit the package availability to a subset of units. The author of a package also has the option of defining packages within other units limiting the scope to teh defining unit. By choosing an appropriate location for a package's definition, the package's author can limit the scope of possible changes.

The final aspect of packages examined in this paper concerns the visibility of packages within other units. A package is visible in a unit if one of the following occurs. First, the package is named in a *with* clause at the beginning of the unit. Second, the package is visible in the unit's parent unit. Items declared in the package can be made directly visible with a *use* clause in the same manner. However, in this paper, we concentrate on general visibility as opposed to direct visibility. Reducing package visibility should lower the vocabulary of the unit. Studies [Elshoff 76] show that this might increase the comprehensibility of the unit. In addition, it would lower the number of possible bindings [Basili and Turner 75] between the unit and the package.

4. Package Metrics

Metrics based on packages can be used to characterize the structure of a program. They can also indicate where problems may occur if the representations of data objects or the implementations of operations change. Most importantly, they provide feedback to programmers about the effects of their definition and use of packages.

There are many simple characterizing metrics that provide a sketch of the system: the number of packages declared, and the number of generic packages and the number of times each is instantiated. In addition to these simple metrics, two more elaborate metrics are discussed below.

4.1. Package Visibility

One means of measuring packages is to look at their visibility to other units in the system. We examine two versions of a system - the current one and one where the *with* clauses may have been moved to lower the visibility.

4.1.1. Definitions

Each package has the following visibility measures:

(1) *used:* number of units where information from the package is accessed or changed.

(2) *current:* number of units where the package is currently visible.

(3) *available:* number of units where the package could be made visible by adding a *with* clause, given the current unit structure.

(4) *proposed:* number of units where the package is visible given the current unit structure and the *with* clauses in their lowest possible positions.

Only those units that are part of the current system are included in our measures. In addition, package bodies and their subunits are not included in the measures for that package because they are part of its implementation. These values can be computed during the design of a system or after the system has been completed.

4.1.2. Examples

Figures 1 and 2 illustrate three views of a small system containing the following components: package P defining a procedure X; and procedure A defining subunits B, C, and D. In Figure 1, there is a *with* clause for P in A; therefore, P is currently visible in four units. (X is not included in these measures because it is a subunit of P.) However, P.X is only used in two units, B and D. In Figure 2, we propose moving the *with* clause from A to B and D, limiting the visibility of P to three units. For a system in which C is a subunit of B, that is the lowest location for the *with* clauses.

4.1.3. Visibility Ratios

Three interesting visibility ratios for the systems in Figures 1 and 2 are given in Table 1.

Table 1: Ratios of Visibility Vectors

Full Name	Notation	(1)	(2)
used/available	UA(P)	0.5	0.5
used/proposed	UP(P)	0.7	1.0
proposed/current	PC(P)	0.8	1.0

Each of these ratios has an upper bound of one and a lower bound of zero. The ratio of units in which a package is accessed to those in which it could be made available (UA) is a measure of the perceived generality of the package. If UA(P) is high, package P is only available where necessary; therefore, the designers may have made it for a specific purpose. However, if UA(P) is low (i.e., P is available much more widely than is necessary), the designers may believe that the package is generally useful.

The ratio of the number of units in which the package is accessed to those where it must be made available (UP) measures excess visibility for the current system structure. For example, in Figure 2, P's visibility in C cannot be eliminated as long as C is a subunit of B. UP can be used to compare different system structures in which excess visibility has been removed.

Finally, the ratio of the number of units in which a package must be visible to those in which it is visible measures the design decision to minimize visibility. If PC(P) is low, P is considered as global data or operations even if a less visible arrangement is available. If PC(P) is high, a decision might have been made to limit the visibility of P.

Had the design goal been minimizing the visibility of packages used, the second subunit structure would be better. The ratios should be used to indicate which packages should be examined more closely, not to replace the need to understand why design decisions have been made.

4.2. Component Accesses

When selection operations are applied to composite objects outside package bodies, details of data representation are spread throughout the program. Distributing representation information rather than centralizing it in private parts of package specifications makes programs more difficult to change. If the type of the composite object is not defined locally, changes in representation to enhance program capability or efficiency could involve many statements in many compilation units.

For example, consider the Ada fragment below containing the visible type T2 having two array components (A and B) with identical numbers of elements with the same type (T).

```
-- original representation
N: constant integer := ...;
type T1 is array (1..N) of T;
type T2 is
  record
    ...;
    A: T1;
    B: T1;
    ...;
  end record;
```

If we introduced a new type, NewType, a record of two elements of type T, that permitted the two array components of the previous example to be combined into a single component (C) in T2:

```
-- new representation
N: constant integer := ...;
type NewType is
  record
    A: T;
    B: T;
  end record;
type T1 is array (1..N) of NewType;
type T2 is
  record
    ...;
    C: T1;
    ...;
  end record;
```

then all references to the A and B components of type T2 variables (e.g., V) would have to be changed.

```
-- original representation     -- new representation
        V.A(...)                    V.C(...).A
        V.B(...)                    V.C(...).B
```

The following measures view component accesses from the user's perspective. The ratio of component accesses to objects with non-locally defined data types to lines of text can be used to measure the code's resistance to changes. The ratio of component accesses to objects with non-package defined data types to lines of text can be used as an indication that more data might be packaged. As in the visibility measures, subunits of a package are considered part of the package; therefore, any accesses in them to the enclosing package are considered local.

Another view of component accesses is from the package's perspective. If components of a package's objects or types are accessed frequently or in many other units, changes to the package may affect all of those units. If the representation is available but never or rarely used, changing it might be easier although encapsulation may not be as necessary. Measures of interest here are the number of component accesses made to objects or types defined by the package and the number of units in which those accesses occur.

5. A Case Study

The visibility and component access metrics described in the previous section have been applied to a subset of an existing ground-support system for a satellite, which was redesigned and implemented in Ada [Basili et al. 82], [Basili et al. 84], [Basili et al. 85]. With the help of the original designers of the system, requirements were developed for a subset system that included an interactive operator interface, graphic output routines, and concurrent telemetry monitoring.

This was an early Ada development to examine the effect of using Ada in an industrial environment. The four programmers had diverse backgrounds. The lead programmer had substantial industrial experience in the application area and was fluent in FORTRAN and assembler languages. The senior programmer had less experience in the application area, but wider exposure to languages (COBOL, PL/I, Lisp, ALGOL, and SNOBOL as well as FORTRAN and assembler). The junior programmer was a recent computer science graduate who had no industrial experience. He was familiar with Pascal in addition to all the languages with which the senior programmer was familiar. The programmer/librarian was a novice programmer who had taken a single course in FORTRAN programming.

Since none of the programmers was familiar with Ada, a one-month training period preceded the start of the project. They viewed fifteen hours of videotaped lectures given by Ichbiah, Firth, and Barnes. A six-day in-house course by a consultant was spread over a period of weeks to allow team members to complete assignments, the last of which was a 500-line team project. Another half-day was spent reviewing the programming practices they were expected to use: design and code walkthroughs and structured programming. The results from this project suggest that this training was not sufficient given the team's background. [Basili et al. 85]

The bulk of the development of this system was done with the Ada/Ed interpreter between February and December 1982. Some testing was done on the ROLM compiler in the summer of 1983. The project was not completed for a number of reasons. The most important of those reasons was the lack of production-quality compilers. The structure of the system as well as the package use can be determined at this stage, however.

5.1. The Program

The final program contained 4375 text lines (excluding comments and blank lines) of Ada. The system contained 11 packages contained in 19 units and a main program with 29 subunits. Some attempts were made to decompose the functions of the subunits; therefore, as many as four nesting levels of subunits are used in the system. Figure 3 shows the structure of the system. Of the eleven packages defined, one's body was not written and it was never used.

The packages were of four types:

(1) 2 common blocks exporting only definitions,

(2) 3 libraries exporting only functions,

(3) 4 encapsulated data types exporting private type definitions and operations, and

(4) 2 data types which exported the representation of the type.

While these numbers seem to indicate that the package feature was used appropriately, closer examination refutes this conclusion. Of the four packages defining encapsulated data types, two were device drivers, another was a mathematical function, and the remaining package definition was neither completed nor used. Device drivers and mathematical libraries are common modules in existing software systems -- no new fully encapsulated types were defined.

Five of the eleven packages were generic, but each was instantiated only once. The generic parameters were primarily ranges for arrays and precision for real numbers. The programmers used the standard sequential_io package in various instantiations, but they only used one instantiation for each of the generic packages they defined. Of those five instantiations, two were in one other package, and three were in the main program. The programmers seemed to view packages as global entities.

A more alarming discovery was that only two of the team members (those with experience in the widest variety of programming languages) defined any packages. The other two team members used the conventional packages provided by the other two programmers (i.e., the device drivers and the mathematical subroutines) and the standard Ada packages but wrote none of their own. Even though the requirements specified that an antenna beam-forming network had binary tree-like connectivity and a binary tree package specification was written, one programmer wrote some different internal functions that manipulated a different representation instead of providing a package body to match the specification.

6. Package Visibility

An examination of the visibility of the packages within the other units indicates that the system structure did not minimize package visibility. The visibility ratios for the 10 packages which were used are given in Table 2.

Table 2: Package Visibility Vectors and Ratios

Package	Used	Proposed	Current	Available	UP	UA	PC
1	9	13	30	44	0.7	0.2	0.4
2	4	9	33	45	0.4	0.1	0.3
3	20	30	32	46	0.7	0.4	0.9
4	1	31	33	47	0.0	0.0	0.9
5	11	30	30	47	0.4	0.2	1.0
6	4	9	30	47	0.4	0.1	0.3
7	7	13	30	47	0.5	0.1	0.4
8	3	3	3	48	1.0	0.0	1.0
9	1	1	2	48	1.0	0.0	0.5
10	1	1	1	48	1.0	0.0	1.0

Although the main program only used two packages, six of the nine packages were named in both *with* and *use* clauses there. Most of the packages were viewed as global data or functions that were accessible everywhere. This view is consistent with the FORTRAN style of programming most familiar to the programmers.

Note that the UA column is fairly low for all the packages except package 3. Package 3 contains type definitions and constants used throughout the system. The low values of UA suggest that some of the packages could be defined locally to groups of units.

The system structure allows reasonable visibility for many of the packages, as indicated in the UP column. It is possible to make packages 8, 9, and 10 visible only in the

packages which use them. Packages 1 and 3 do not have an inordinate amount of visibility. However, packages 2, 4, 5, 6, and 7 would probably benefit from a change in the subunit structure if it were possible.

Finally, the PC column demonstrates the programmers' view of the role of packages in the system. Five of the packages were given the appropriate visibility in the original system; however, the rest of the packages are visible much more than they need to be. This is yet another example of the programmers' global data approach to package definition.

7. Component Accesses

Table 3 summarizes non-local component accesses for each programmer based on all modules written by the programmer. The total number of component accesses, the accesses to packaged data, and the accesses to non-packaged data are each included.

These metrics show that on average more than one of every ten lines (0.11) of text contained a reference to a component of an object with an externally defined type. Roughly twice as many references are made to packaged data as are made to unpackaged data which suggests that the more complex data types might have been packaged but not hidden. However, programmer 3 made more references to components of unpackaged data.

Table 4 summarizes the component accesses by package for selected packages. Note that package 3 has 217 of the packaged component accesses. This is not surprising considering that the package contains global data and types. The majority of the remaining packaged component accesses were to package 7 which provides some types shared by several related units. If the data in these two packages were hidden, the number of packaged component accesses and the effect of changes to the packages would be greatly diminished. However, changes to the representation at this time would affect many other units.

Table 3: Component Accesses by Programmer

Metric	Programmer				
	1	2	3	4	Total
Text Lines	706	1904	1648	117	4375
Component Accesses					
all data	159	171	140	0	470
only packaged data	120	124	61	0	305
excluding packaged data	39	47	79	0	165
Accesses per Text Line					
all data	0.23	0.09	0.08	0.00	0.11
only packaged data	0.17	0.06	0.04	0.00	0.07
excluding packaged data	0.06	0.02	0.05	0.00	0.03

Table 4: Component Accesses by Package

Package	# of Units	# of Accesses
3	13	217
2	1	9
8	2	14
7	6	46
5	1	19

The values in Table 3 indicate that the first programmer's code should be relatively difficult to change since about one of every five lines contained a component access. We selected one of this programmer's modules and made the trivial modification discussed in Section 4.2. To make this change in the representation, eleven program changes [Dunsmore and Gannon 77] were required in the module we selected. In addition, ten program changes were needed in five other modules which encompassed two of the four major subsystems in the program. The record type containing these components could have been encapsulated in a package definition. Then, the same change in the representation would require a change in the private part of the package specifications and a total of four program changes in two functions of the package body. No other modules would be affected.

8. Conclusions

The case study demonstrates what might happen when programmers who are experienced in an application area but lack training in modern software development practices begin to use Ada. Despite training efforts that are similar to those that are likely to be used in a typical industrial setting, only traditional modules like device drivers and mathematical libraries were defined. Encapsulated types were declared only by programmers with the widest exposure to different languages, but even the programmers' prior success in working with these languages does not guarantee success with Ada. A good background in the software engineering practices which Ada supports is probably necessary to learn to use the full capabilities of the features of the language -- simply teaching professional programmers Ada is not enough.

Had the package metrics been applied during the case study, they might have helped the programmers better understand how to use packages. Package visibility is a rather crude metric that can be used during design to check that the designer's approach to a system is not simply to make all packages visible to all program units. Lowering the visibility will probably decrease the scope of any changes made to the package.

However, even if package visibility is restricted, packages may still export type definitions that permit programmers to access the components of composite objects. Program units that directly access components of objects are likely to be difficult to change.

Metrics which track the use of packages during system development treat the symptoms and not the problem; however, we expect many early developments will have these symptoms. These metrics and those described in [Hammons and Dobbs 85] may help in the transition to using Ada effectively.

9. Acknowledgements

M.V. Zelkowitz, J.B. Bailey, E. Kruesi Bailey, and S.B. Sheppard were the other monitors of the case study and have contributed to the work reported here.

10. References

[Basili and Turner 75]
V.R. Basili and A.J. Turner. Iterative Enhancement: A Practical Technique for Software Development. *IEEE Trans. on Software Eng.* SE-1, 4, 1975, pp. 390-396.

[Basili et al. 82]
V. Basili, J. Gannon, E. Katz, M. Zelkowitz, J. Bailey, E. Kruesi, and S. Sheppard. Monitoring an Ada software development project. Ada Letters II, 1, (July 1982), 1.58-1.61.

[Basili et al. 84]
V. Basili, S. Chang, J. Gannon, C. Loggia-Ramsey, E. Katz, N. Panilllo-Yap, M. Zelkowitz, J. Bailey, E. Kruesi, and S. Sheppard. Monitoring an Ada software development project. Ada Letters IV, I, (July/August 1984) 32-39.

[Basili et al. 85]
V.R. Basili, E.E. Katz, N.M. Panilllo-Yap, C.L. Ramsey, and S. Chang. Characterization of a Software Development in Ada. *IEEE Computer*. Vol. 18, No. 9, Sept. 1985, pp. 53-65.

[Dunsmore and Gannon 77]
H.E. Dunsmore and J.D. Gannon. Experimental investigation of programming complexity. Proceedings of the ACM/NBS 16th Annual Technical Symposium, Gaithersburg, Md., (June 1977), 117-125.

[Elshoff 76]
J.L. Elshoff, An analysis of some commercial PL/1 programs. *IEEE Trans. on Software Eng.* SE-2, 2, 1976, pp. 113-120.

[Gannon and Horning 75]
J.D. Gannon and J.J. Horning. Language design for programming reliability. *IEEE Trans. on Software Eng.* SE-1, 2, 1975, pp. 179-191.

[Hammons and Dobbs 75]
C. Hammons and P. Dobbs. Coupling, cohesion, and Package Unity in Ada. Ada Letters IV, 6, (May-June 1985) pp. 49-59.

[Ichbiah et al. 79]
J.D. Ichbiah, J.G.P. Barnes, J.C. Hellard, B. Krieg-Bruckner, O. Roubine, and B.A. Wichman. Rationale for the design of the Ada programming language. SIGPLAN Notices 14, 6, (June 1979), 8-1,2.

[Ledgard 85]
H.L. Ledgard. Packages: a method for software decomposition. 1985.

[Parnas 71]
D.L. Parnas. Information distribution aspects of design methodology. IFIP 71, North Holland Pub. Co., Amsterdam, (1971), 339-344.

```
package P is
  procedure X;
end P;
--------------
separate (P)
procedure X is
begin
  ...;
end X;
--------------
with P;
procedure A is
  procedure B is separate;
  procedure D is separate;
begin
  ...;
end A;
```

```
separate(A)
procedure B is
procedure C is separate;
begin
  ...;
  P.X;
  ...;
end B;
--------------
separate (A.B)
procedure C is
begin
  ...;
end C;
--------------
separate (A)
procedure D is
begin
  ...;
  P.X;
  ...;
end D;
```

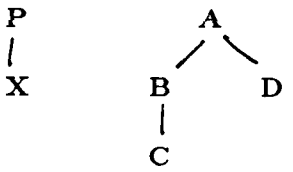


FIGURE 1: Global Visibility of Package P

```
package P is
  procedure X;
end P;
----------
separate (P)
procedure X is
begin
  ...;
end X;
----------
procedure A is
  procedure B is separate;
  procedure D is separate;
begin
  ...;
end A;
```

```
with P;
separate(A)
procedure B is
procedure C is separate;
begin
  ...;
  P.X;
  ...;
end B;
----------
separate (A.B)
procedure C is
begin
  ...;
end C;
----------
with P;
separate (A)
procedure D is
begin
  ...;
  P.X;
  ...;
end D;
```

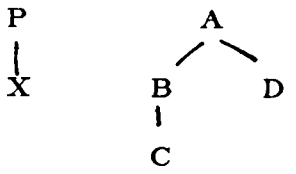
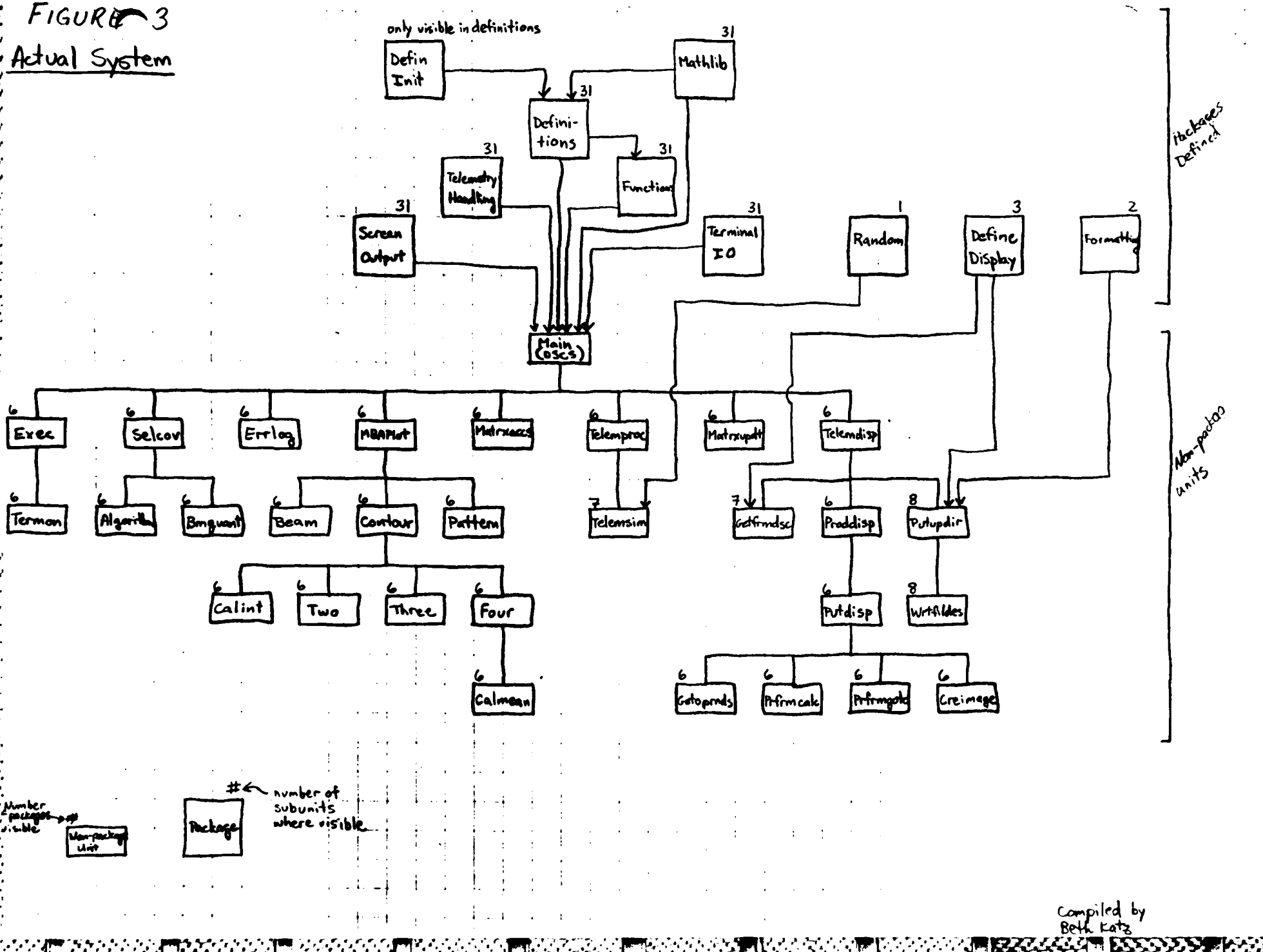


FIGURE 2: Limited Visibility of Package P

FIGURE 3

Actual System



Compiled by
Beth Katz

END
FILMED
4-86
DTIC